II Chart Parsing

Les algorithmes de la partie I posent deux problèmes :

1. Ils posent des contraintes pour les grammaires qu'on peut utiliser.
2. Ils ne sont pas efficaces parce que souvent, on fait plusieures fois la même analyse pour une partie de la phrase.

Exemple 16

Analyse de la phrase

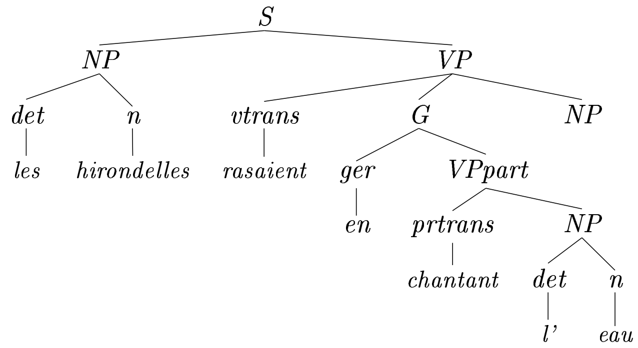
les hirondelles rasaient en chantant l'eau

Productions :

$S \to NP\ VP$	$NP \to det\ n$	$VP \to vtrans\ G\ NP$
$G \to ger\ VPpart$	$VPpart \to prtrans\ NP$	$VPpart \to printrans$

Entrées lexicales : $chantant \in prtrans$ $chantant \in printrans$ $rasaient \in vtrans$ …

Analyse top-down mène d'abord à l'arbre suivant :



Après on fait un backtracking en annulant tout jusqu'à l'application de VPpart → prtrans NP.

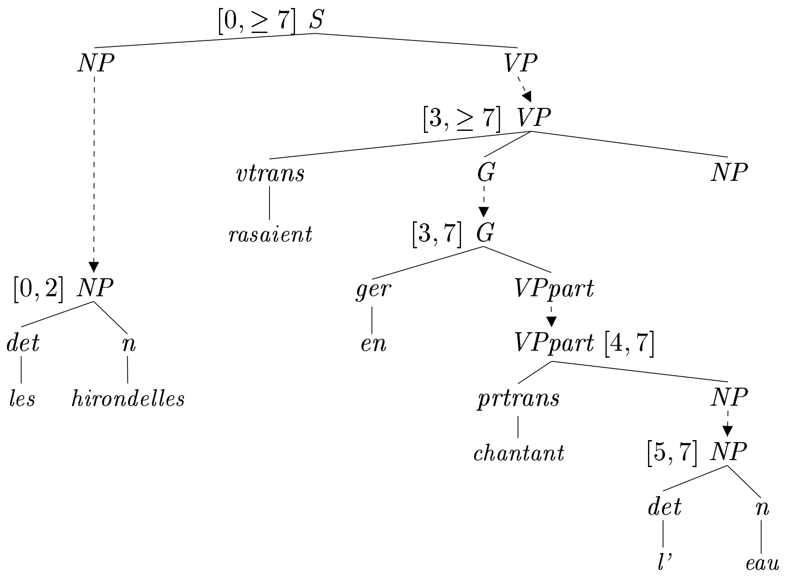
Par conséquent, le NP 'l'eau' est analysé deux fois.

Un cas encore pire : *Jean mange un sandwich et ensuite il s'en va* avec productions S → NP VP et S → S coord S : la première phrase est analysée deux fois.

Idée du chart parsing : les analyses des parties de la phrase sont gardées jusqu'à la fin, même si une telle analyse est utilisée comme partie d'une analyse plus grande.

Exemple 17

Au moment du backtracking dans l'exemple précédant, on a les analyses (partielles ou complètes) suivantes :



$[i, j]$ indique que l'analyse est une analyse complète qui couvre la partie de la phrase entre positions i et j. $[i, \geq j]$ indique que l'analyse est une analyse partielle couvrant la partie de la phrase entre positions i et j et que plus tard elle pourrait même aller jusqu'à une position plus grande que j.

Quand on refait l'analyse du VPpart, on garde l'analyse du NP 'l'eau' et on la réutilise plus tard.

Chart = structure de donnée qui contient les sous-analyses (partielles ou complètes) déjà trouvées.

Plusieures réalisations possibles, souvent comme graphe acyclique, vector ou matrice.

Les éléments d'un chart s'appellent *arcs*. On distingue

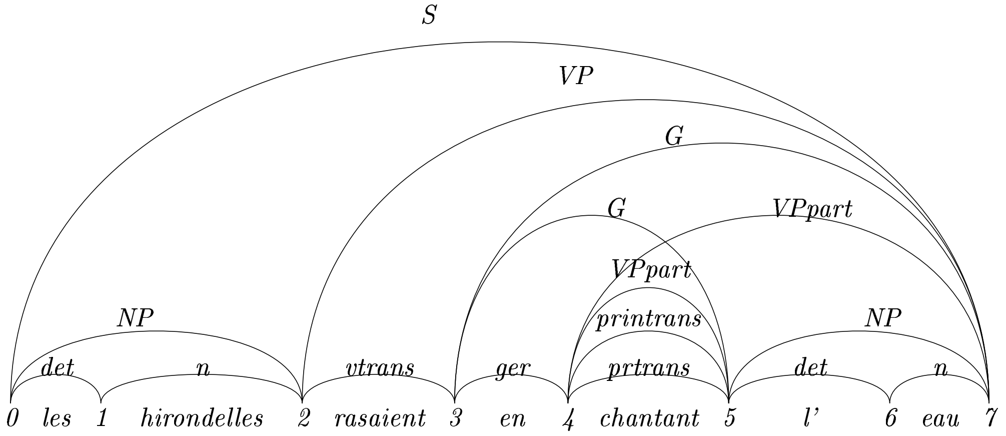
- des arcs actifs = analyses partielles, et
- des arcs passifs = analyses complètes.

Un arc contient à moins les informations suivantes :

1. la catégorie de l'analyse (du sous-arbre)
2. une spécification de la partie de la phrase couverte par cette anlyse
3. dans le cas d'un arc actif : une specification de la partie déjà reconnue et de la partie encore à reconnaître

Pour spécifier des parties d'une phrase, on utilise des positions $0, \ldots, n$ entre les mots d'une phrase $w_1 \ldots w_n$: $0\ w_1\ 1\ w_2\ 2\ w_3\ \ldots\ n-1\ w_n\ n$. La partie entre positions i et j consiste en les mots $w_{i+1} \ldots w_j$.

Exemple 18 *Représentation d'un chart avec seulement des arcs passifs comme graphe acyclique : Considérons l'exemple prédédant.*



Il existe un arc avec S de 0 à n ⇒ analyse trouvée.

Exemple 19 *Même exemple, représentation en vector :*

Le vector v contient n champs. On met une entrée $[A, i]$ dans le champ v_j si on a trouvé une analyse avec catégorie A couvrant la partie entre positions i et j.

v_1	$[det, 0]$
v_2	$[n, 1]$ $[NP, 0]$
v_3	$[vtrans, 2]$
v_4	$[ger, 3]$
v_5	$[prtrans, 4]$ $[printrans, 4]$ $[VPpart, 4]$ $[G, 3]$
v_6	$[det, 5]$
v_7	$[n, 6]$ $[NP, 5]$ $[VPpart, 4]$ $[G, 3]$ $[VP, 2]$ $[S, 0]$

Dans v_n il y a une entrée $[S, 0]$ ⇒ analyse trouvée.

Exemple 20 *Même exemple, représentation en matrice M :*

Le champ $m_{i,j}$ contient une catégorie A si on a trouvé une analyse avec catégorie A couvrant la partie entre positions i et j.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0		*det*	*NP*					*S*
1			*n*					
2				*vtrans*				*VP*
3					*ger*	*G*		*G*
4						*prtrans* *printrans* *VPpart*		*VPpart*
5							*det*	*NP*
6								*n*
7								

Dans $M_{0,n}$ il y a une entrée S ⇒ analyse trouvée.

5 Chart Parsing bottom-up

Dans cette partie, on va considérer deux algorithmes : un algorithme de chart parsing bottom-up général et l'algorithme de Cocke-Kasami-Younger.

Pour tous les deux on a les contraintes suivantes pour les grammaires (comme dans le cas du parsing bottom-up comme décrit dans la partie 3 du cours) :

- pas de productions de la forme A $\rightarrow \epsilon$, $A \in V$,
- pas de cycles A $\stackrel{n}{\Rightarrow}$ A, $n \geq 1$, $A \in V$.

5.1 Un algorithme général de chart parsing bottom-up

Idée :

- d'abord trouver tous les arcs lexicaux
- ensuite en construire des arcs syntaxiques

Notation : $[i, A, j]$: arc avec catégorie A couvrant la partie entre positions i et j.

Procédure arcs-lex

```
Données globales :  dictionnaire D
Input :  mot m, index i
Output :  ensemble d'arcs lexicaux
Variable locale :  l-arcs :  ensemble d'arcs, valeur intiale ∅
Méthode :
   for all ⟨x,t⟩ from D :
      if x = m
      then l-arcs := l-arcs ∪ {[i − 1, t, i]}
   return(l-arcs)
```

Pour trouver des arcs syntaxiques à partir d'un ensemble d'arcs déjà trouvé, une procédure `arcs-synt` fait le suivant :

- elle choisit un arc $[i, X_1, j_1]$
- pour toutes les productions $X \rightarrow X_1 X_2 \ldots X_k$:
  si on a déjà trouvé des arcs $[j_1, X_2, j_2], \ldots, [j_{k-1}, X_k, j_k]$, on peut ajouter $[i, X, j_k]$.

Procédure arcs-synt

```
Données globales :  CFG G = ⟨V,T,S,P⟩
Input :  ensemble d'arcs chart
Output :  ensemble d'arcs syntaxiques
Variable locale :  s-arcs :  ensemble d'arcs, valeur intiale ∅
Méthode :
   for all [i,X,j1] from chart :
      for all X → X1 X2 ... Xk from P
         if il existe j2,...,jk tel que
            [j1,X2,j2],...,[jk−1,Xk,jk] ∈ chart ∧ [i,X,jk] ∉ chart
         then s-arcs := s-arcs ∪ {[i,X,jk]}
   return(s-arcs)
```

Algorithme chart-recognize

`bottom-up/gauche-droite/breadth-first`

```
Données globales :  CFG G = ⟨V,T,S,P⟩, dictionnaire D
Input :  w = w_1 ... w_n ,  n ≥ 1
Output :  true/false
Variable locale :  chart :  ensemble d'arcs, valeur intiale ∅,
      new :  ensemble d'arcs
Méthode :
   for all i from {1,...,n} :
      chart := chart ∪ arcs-lex(w_i, i)
   while [0,S,n] ∉ chart
      new := arcs-synt(chart)
      if new = ∅
      then return(false)
      else chart := chart ∪ new
   return(true)
```

Exemple 21 *Reprenons l'exemple de l'introduction à la partie II du cours.*

Après les appels de `arcs-lex` *:*

$chart = \{[0,Det,1],[1,N,2],[2,vtrans,3],[3,ger,4],[4,prtrans,5],[4,printrans,5],[5,Det,6],[6,N,7]\}$

Ensuite (dans le `while`*) on ajoute à chart*

1. *premier boucle :* $[0,NP,2],[5,NP,7],[4,VPpart,5]$
2. *deuxième boucle :* $[3,G,5],[4,VPpart,7]$
3. *troisième boucle :* $[3,G,7],[2,VP,7]$
4. *quatrième boucle :* $[0,S,7]$

Problème avec cet algorithme : la calculation de *chart* demande beaucoup de comparaisons.

5.2 L'algorithme de Cocke, Kasami et Younger

L'algorithme de Cocke, Kasami et Younger est un cas spécial de l'algorithme présenté dans 5.1.

L'algorithme de Cocke, Kasami et Younger suppose que la grammaire est sous forme normale de Chomsky, c'est-à-dire toutes les productions ont une des deux formes suivantes :

- $A \to BC$ avec $B, C \in V$ ou
- $A \to t$ avec $t \in T$.

Beaucoup de grammaires pour langages naturels adoptent des arbres binaires. Pour elles, la contrainte de la forme normale de Chomsky ne pose pas de problèmes.

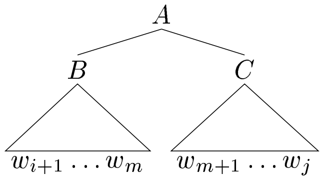
En général, pour chaque CFG sans des productions de la forme $A \to \epsilon$ on peut construire une grammaire faiblement équivalente sous forme normale de Chomsky.

Avec une grammaire sous forme normale de Chomsky, pour analyser une phrase $w = w_1 w_2 \ldots w_n$ on doit appliquer n productions de la forme $A \to t$ et $n-1$ productions de la forme $A \to BC$, parce que chaque production de la dernière forme augmente le nombre de non-terminaux par 1.

Le chart est représenté comme une matrice *chart*. Pour une phrase de longueur n on a seulement besoin des champs $chart_{i,j}$ avec $0 \leq i < j \leq n$.

Idée de l'algorithme : on calcule d'abord les préterminaux (considéré comme non-terminaux), ensuite on calcule les regroupement de $k = 2, 3, ..., n$ symboles.

Comme on sait que toutes les productions pour construire des structures syntaxiques sont de la forme $A \to BC$, pour trouver une catégorie A couvrant $w_{i+1} \ldots w_j$ il suffit de trouver la position m qui sépare les deux parties qui correpondent aux catégories B et C dans une production $A \to BC$. Si, pour une telle production, un tel m existe, on peut ajouter A dans $chart_{i,j}$.



Algorithme recognize-cky

`bottom-up/breadth-first`

Données globales : CFG $G = \langle V, T, S, P \rangle$ sous forme normale de Chomsky
Input : phrase $w = w_1 \dots w_n$
Output : true/false
Variable locale : *chart* : matrice $n \times n$, position j
Méthode :
   for all i from $\{1, \dots, n\}$:
      $chart_{i-1,i} := \{X | X \to w_i \in P\}$
   for all k from $\{2, \dots, n\}$:
      for all i from $\{0, \dots, n-k\}$:
         $j := i + k$
         $chart_{i,j} := \{A \,|$ il existe un $A \to BC \in P$ et un $m, i < m < j$ tel que
                  $B \in chart_{i,m} \;\wedge\; C \in chart_{m,j}\}$
   if $S \in chart_{0,n}$
   then return(true)
   else return(false)

Une variante qui calcule un chart plus complexe (contenant des sous-arbres de hauteur 1 au lieu de la catégorie seulement) peut être obtenue comme suit :

On remplace dans l'algorithme la partie calculant $chart_{i,j}$ à partir des productions par :

   for all k from $\{2, \dots, n\}$:
      for all i from $\{0, \dots, n-k\}$:
         $j := i + k$
         $chart_{i,j} := \{\langle A, (B, i, m), (C, m, j) \rangle \,|$ il existe un $A \to BC \in P$
                  et un $m, i < m < j$ tel que
                  $\langle B, \beta_1, \beta_2 \rangle \in chart_{i,m}$ pour quelques β_1, β_2
                  $\wedge\; \langle C, \gamma_1, \gamma_2 \rangle \in chart_{m,j}$ pour quelques $\gamma_1, \gamma_2\}$

On accèpte dans ce cas si on trouve une entrée $\langle S, \alpha, \beta \rangle$ pour quelque α et β dans $chart_{0,n}$.

Exemple 22

Productions de la forme $A \to BC$:

$S \to NP\ VP \quad NP \to Det\ N \quad N \to Adj\ N \quad VP \to V\ NP$

Productions de la forme $A \to a$:

Det → le	*Adj → petit*	*Adj → grand*
N → chien	*N → chat*	*v → chasse*

le petit chien chasse le grand chat

	1	2	3	4	5	6	7
0	*Det*		*⟨NP, (Det, 0, 1), (N, 1, 3)⟩*				*⟨S, (NP, 0, 3), (VP, 3, 7)⟩*
1		*Adj*	*⟨N, (Adj, 1, 2), (N, 2, 3)⟩*				
2			*N*				
3				*V*			*⟨VP, (V, 3, 4), (NP, 4, 7)⟩*
4					*Det*		*⟨NP, (Det, 4, 5), (N, 5, 7)⟩*
5						*Adj*	*⟨N, (Adj, 5, 6), (N, 6, 7)⟩*
6							*N*

Pour un tel chart, c'est relativement facile d'obtenir une analyse (une arbre de dérivation). Ceci est fait par la procédure `structure-cky`. `structure-cky` prend trois arguments : la catégorie A est deux positions i, j. Initialement, `structure-cky` est appelé avec S, 0 est n comme arguments.

Procédure structure-cky

`non-déterministe`

```
Données globales :  phrase w_1...w_n et chart chart pour cette phrase
Input :  Symbole A, position i,j avec 1 ≤ i < j ≤ n
Output :  analyse
Méthode :
   if i = j − 1
   then return([A w_j])
   else
      choisit un ⟨A,(B,i,m),(C,m,j)⟩ ∈ chart_{i,j}
      return([A structure-cky(B, i, m) structure-cky(C, m, j)])
```

Dans les deux algorithmes considérés dans 5.1 et 5.2, on n'a mis que des arcs passifs dans le chart.

Compléxité d'un algorithme :

1. son besoin de temps, et
2. son besoin de mémoire

On ne peut pas donner des caractérisation en une mésure précise, des secondes par exemple, parce que le besoin de temps et de mémoire d'un programme (d'une réalisation de l'algorithme) dépendent

- des données entrant dans le programme (surtout de leur taille)
- du complexité de l'algorithme utilisé
- du langage de programmation et du compilateur utilisé
- de la nature et la vitesse d'exécution des instructions du microprocesseur utilisé

Quand il s'agit de l'algorithme, on ne connaît pas les deux derniers aspects et la taille des données d'entrée est variable.

La complexité d'un algorithme est spécifié en fonction de la taille des données d'entrée, dans notre cas la longueur n de la phrase à analyser (et parfois la taille de la grammaire).

On peut considérer le besoin de temps/mémoire dans le pire des cas (*worst case*), le cas moyen ou le meilleur cas. Pour caractériser la complexité, on considère en général le pire des cas.

On ne peut pas donner une spécification précise mais plutôt quelque chose comme "le besoin de temps/mémoire est plus petit ou égal à quelque chose qui est proportionnel à $f(n)$" avec n la longueur de la phrase, f une fonction. Dans ce cas on dit aussi "l'algorithme est d'une complexité de temps/mémoire de $\mathrm{O}(f(n))$.

Définition 5 *g et f soient deux fonctions.*

$g(n) = O(f(n))$ ssi il existent deux constantes c et n_0 tel que pour tous les $n \geq n_0$: $g(n) \leq cf(n)$.

Theorème 1 (Complexité de `recognize-cky`) *G soit une grammaire hors contexte sous forme normale de Chomsky. Pour une phrase w avec $|w| = n$,*

1. *le besoin de temps pour calculer* `recognize-cky`*(w) en utilisant G est $O(n^3)$.*
2. *le besoin de mémoire pour calculer* `recognize-cky`*(w) en utilisant G est $O(n^2)$.*

Complexité de temps : Considérons la première version de `recognize-cky`.

`for all` i `from` $\{1, \ldots, n\}$ `:`
$chart_{i-1,i} := \{X | X \rightarrow w_i \in P\}$

$\Rightarrow$ n opérations dont chacune avec un besoin de temps délimité par une constante $c_1 \Rightarrow c_1 n$

`for all` k `from` $\{2, \ldots, n\}$ `:`

`for all` i `from` $\{0, \ldots, n-k\}$ `:`

$\Rightarrow$ $n - 1$ opérations dont chacune consiste en $n - k + 1 \leq n - 1$ opérations

$j := i + k$

$\Rightarrow$ opération avec besoin de temps $\leq c_2$ pour quelque constante c_2

$chart_{i,j} := \{A \,|$ `il existe un` $A \rightarrow BC \in P$ `et un` $m, i < m < j$ `tel que`
$B \in chart_{i,m} \;\wedge\; C \in chart_{m,j}\}$

$\Rightarrow$ pour chaque $m, 1 \leq i < m < j \leq n$ il faut vérifier s'il existe un $A \rightarrow BC$ et des entrées correspondantes dans *chart*. Le temps pour chercher la production est délimité par une constante c_3, et comme on a au plus $|V|$ éléments dans un champ, le temps nécessaire pour la recherche dans un champ du *chart* et délimité par une constante c_4

$\Rightarrow$ besoin de temps $\leq (j - i - 1)(c_3 + 2c_4) \leq n(c_3 + 2c_4)$

$\Rightarrow$ besoin de temps pour le grand boucle :
$\leq (n-1)(n-1)(c_2 + n(c_3 + 2c_4)) \leq c_5 n^3$ pour quelque constante c_5

`if` $S \in chart_{0,n}$
`then return(true)`
`else return(false)`

$\Rightarrow$ besoin de temps pour ce dernier `if then else` : délimité par une constante c_6

$\Rightarrow$ besoin de temps pour tout l'algorithme :
$\leq c_1 n + c_5 n^3 + c_6 \leq c_7 n^3$ pour quelque constante c_7.

Besoin de mémoire : On a besoin d'une matrice avec $n \times n = n^2$ champs dont les champs ont un besoin de mémoire constant.

La deuxième version de `recognize-cky` est plus complexe parce que la taille d'un champ de la matrice est $\mathrm{O}(n)$. Mais on peut aussi obtenir une analyse à partir de la première version en $\mathrm{O}(n^2)$. Pour obtenir un résultat sur la complexité de l'algorithme de Cocke, Kasami et Younger, c'est alors justifié de considérer seulement la première version de `recognize-cky`.